Low Latency Event-Based Filtering and Feature Extraction for Dynamic Vision Sensors in Real-Time FPGA Applications

Linares-Barranco, Alejandro ; Perez-Pena, Fernando ; Moeys, Diederik Paul ; Gomez-Rodriguez, Francisco ; Jimenez-Moreno, Gabriel ; Liu, Shih-Chii ; Delbruck, Tobi

Abstract: Dynamic Vision Sensor (DVS) pixels produce an asynchronous variable-rate address-event output that represents brightness changes at the pixel. Since these sensors produce frame-free output, they are ideal for real-time dynamic vision applications with real-time latency and power system constraints. Event-based filtering algorithms have been proposed to post-process the asynchronous event output to reduce sensor noise, extract low level features, and track objects, among others. These postprocessing algorithms help to increase the performance and accuracy of further processing for tasks such as classification using spike-based learning (ie. ConvNets), stereo vision, and visually-servoed robots, etc. This paper presents an FPGA-based library of these postprocessing event-based algorithms with implementation details; specifically background activity (noise) filtering, pixel masking, object motion detection and object tracking. The latencies of these filters on the Field Programmable Gate Array (FPGA) platform are below 300ns with an average latency reduction of 188% (maximum of 570%) over the software versions running on a desktop PC CPU. This open-source event-based filter IP library for FPGA has been tested on two different platforms and scenarios using different synthesis and implementation tools for Lattice and Xilinx vendors.

DOI: https://doi.org/10.1109/access.2019.2941282

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 12, 2019, accepted September 2, 2019, date of publication September 13, 2019,
date of current version September 30, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2941282

Low Latency Event-Based Filtering and Feature Extraction for Dynamic Vision Sensors in Real-Time FPGA Applications

ALEJANDRO LINARES-BARRANCO[1], (Senior Member, IEEE), FERNANDO PEREZ-PEÑA[2], (Member, IEEE), DIEDERIK PAUL MOEYS[3], (Member, IEEE), FRANCISCO GOMEZ-RODRIGUEZ[1], (Member, IEEE), GABRIEL JIMENEZ-MORENO[1], (Member, IEEE), SHIH-CHII LIU[3], (Senior Member, IEEE), AND TOBI DELBRUCK[3], (Fellow, IEEE)

[1]Robotic and Technology of Computers Laboratory, University of Seville, 41012 Seville, Spain
[2]Applied Robotics Laboratory, School of Engineering, University of Cadiz, 11519 Puerto Real, Spain
[3]Institute of Neuroinformatics, University of Zurich and ETH Zurich, 8057 Zürich, Switzerland

Corresponding author: Alejandro Linares-Barranco (alinares@atc.us.es)

This work was supported in part by the EU Project VISUALISE, and in part by the Spanish Government Project COFNET under Grant TEC2016-77785-P through the European Regional Development Fund. The work of D. P. Moeys was supported by the EU 7th Framework FP7-ICT-2011-9 Project VISUALISE under Grant 600954.

ABSTRACT Dynamic Vision Sensor (DVS) pixels produce an asynchronous variable-rate address-event output that represents brightness changes at the pixel. Since these sensors produce frame-free output, they are ideal for real-time dynamic vision applications with real-time latency and power system constraints. Event-based filtering algorithms have been proposed to post-process the asynchronous event output to reduce sensor noise, extract low level features, and track objects, among others. These postprocessing algorithms help to increase the performance and accuracy of further processing for tasks such as classification using spike-based learning (ie. ConvNets), stereo vision, and visually-servoed robots, etc. This paper presents an FPGA-based library of these postprocessing event-based algorithms with implementation details; specifically background activity (noise) filtering, pixel masking, object motion detection and object tracking. The latencies of these filters on the Field Programmable Gate Array (FPGA) platform are below 300ns with an average latency reduction of 188% (maximum of 570%) over the software versions running on a desktop PC CPU. This open-source event-based filter IP library for FPGA has been tested on two different platforms and scenarios using different synthesis and implementation tools for Lattice and Xilinx vendors.

INDEX TERMS Neuromorphic engineering, address-event-representation (AER), dynamic vision, frame-free vision, event-based processing, event-based filters, field programmable gate arrays (FPGA), VHDL.

I. INTRODUCTION

Dynamic Vision Sensors (DVSs) [1], [2] mimic part of the biological retina's functionality in silicon chips using an asynchronous output representation called Address Event Representation (AER) [3], [4]. Each sensing unit, or pixel in these vision sensors, models simple ON and OFF retinal ganglion cells. The main advantages of these vision sensors are the sparseness of their visual information, their low-latency response, and their high dynamic range. A sensed log-intensity brightness change by any of the pixels is sent out in typically less than 1*ms* after it is produced. This architecture is radically different to frame-based cameras used in artificial vision. Conventional cameras measure the intensity over a short period of time (exposure time) in all the pixels, and then they then send out the entire frame. This frame transfer is done even though in many scenarios, only a few pixels have changed since the last captured frame.

The biological retina is the only source of visual information to the brain. Visual processing begins when photons stimulate the light-sensitive photoreceptor rod and cone cells in the retina. These cells convert the information into electrical signals and send them through intermediate networked layers of cells to around 15-20 types of retinal ganglion cells.

The associate editor coordinating the review of this manuscript and approving it for publication was Mostafa Rahimi Azghadi.

They perform visual processing before visual information arrives to the visual cortex in the brain. The DVS pixel implements a simplified model of the ON and OFF transient ganglion cells. Other ganglion cell functionality, such as the local and global motion detection and approach responses, aer implemented in both software (jAER [5]) and hardware within the EU VISUALISE project [6].

In the central nervous system, high priority information arrives first to the brain, so it is processed in a higher priority, or even in an involuntary and reflexive way. The use of digital cameras to emulate such kind of processing demands an enormous amount of computational resources to extract that crucial information from the frames in the available interframe time. For hard real-time systems, this biological mimicking approach is valuable and results in visual embedded systems that are able to perform relatively complex visual tasks, like fast object detection and tracking, as we demonstrate in this work.

This paper presents a set of event-based filters, with improvements and extensions over previous works, that are designed to remove uninformative events (for e.g. background noise events) or useless ones (mask filter), and feature extractor algorithms (objects tracking, motion detection). The aim of this work is to serve as a DVS post-processing mechanism for reducing latencies and increasing the accuracy of any event-based processor (i.e. classifier, learning algorithm, etc). We present digital architectures, with implementation details, using hardware description languages for FPGAs. We evaluate these architectures on two different platforms so as to demonstrate their independence towards a particular FPGA vendor or embedded architecture. The library is open-sourced[1] to enable researchers to build on this work by forking.

The paper is structured as follows: Section II presents the related works, then Section III describes the basis of the event-based neuromorphic processing and examples of filters and feature extractors that process the sensor output. Section IV gives the FPGA implementation details for these algorithms. Section V describes the two FPGA platforms we used, and Section VI shows the results.

II. RELATED WORK

Event-based algorithms for DVS have been reviewed in [7]–[10]. Most of the object tracking proposed works [11]–[15] are tested in software for desktop CPUs. In [11], a cluster tracker is used to detect balls in real-time; [12] extends the Fischler and Elschlager model to the event-based domain; a corner detection event-based algorithm is presented in [14]; in [15] the used method is the mean shift as gradient descend for a robotic application in real-time; authors on [13] used a mixed alternative where the object is detected from a frame, and then it is followed in an event-based manner during the inter-frame time, which is valid only for DAVIS retina [2].

Other visual filters have been implemented in software [16] (gaussian, bilateral and Canny edge detector), ASICs [17], FPGAs [18], [19] or a combination of these two last; like [20], where a mixed signal ASIC design implements a correlation filter for DVS output events and it tells if that event must be filtered or not, delegating that job to an FPGA. Recently [21] presented an interesting DVS noise filter that uses only linear arrays of memory for a 2D pixel array; it reduces memory resources up to 10× less, but can only filter out noise at low activity rates.

Reference [22] presents the first event-based, energy-efficient approach for object detection and categorization using the DAVIS [23], called PCA-RECT. This framework requires a training phase in order to perform a kind of pattern matching. It has been tested on FPGA and it has an accuracy of 79% , which might be improved by DAVIS postprocessing to reduce and clean the stream of events from noisy or not correlated events.

Another event-based system that requires training phases to accomplish the object-tracking problem is [24], which offers promising results for several event-based algorithms such as tracking or feature matching. The filters and algorithms presented in this paper achieve a throughput performance of only 59 keps[2] running on an Intel Core i7 as C++ single-thread implementation, which could be considerably improved on FPGA (such low event rates are only observed in staring surveillance scenarios).

Other filters or event-based processing algorithms implemented in hardware to improve their latencies are: (1) real-time frequency detectors [25] for quadcopter motors with low latency, where implementations are for an FPGA at 100MHz and simulated on an ASIC at 155MHz; and (2) Padala *et al.* [26], where authors present a neural network-based noise filtering for TrueNorth [27] with similar functionality of the background activity filter presented in this paper. Its main advantage is the capacity for event generation, but the millisecond resolution presented, although is enough for their application (traffic monitoring), it might be reduced for other high-speed real-time applications.

The filters and features extractors presented in this paper: (1) does not require training from datasets, although the mask filter requires a short observation time from the sensor; (2) they have been tested both in software and FPGA to highlight the latency improvements for FPGAs, their simple components requirements and their viability for high-speed visual applications; and (3), these algorithms belong to those sensor post-processing event-based mechanishms that could improve capabilities of more complex tasks.

III. NEUROMORPHIC EVENT-BASED VISUAL PROCESSING

Event-based processing refers to the processing of those information codified in events produced by neuromorphic sensors. This processing is done before the sensor events are

[1] https://github.com/RTC-research-group/EDIP_library

[2] *eps is events per second, keps is a thousand eps*

transmitted to high level algorithms such as ConvNets [28], Deep Belief Networks [29] for classification tasks, or for robotic applications [30]–[32]. This processing should preserve the low latency property of the sensor output, which comes about because of the asynchronous and quick readout of the sensors such as the DVS. There are two main classes of event-based sensor processing algorithms: filters and feature extractors [7]. *Filters* cannot transform sensor information; they can only apply small changes for improving the quality of the signal, e.g. removing noise events or reducing the activity. On the other hand, *feature extractors* extract a particular stimulus property such as edge orientation. Events can be transmitted with additional information such as the best feature at a pixel.

Let us suppose $E = \{e_0, e_1, e_2, \ldots, e_n\}$ to be a set of events coming out from the DVS sensor. Each of these events is composed of four terms: $e_i = (s_i, x_i, y_i, t_i)$, corresponding to the polarity, x-address, y-address and timestamp of the event respectively. Each filter algorithm is expressed as in (1).

$$E_{FILTER} = f_{FILTER}(E) = \{e_{F_0}, e_{F_1}, e_{F_2}, \ldots, e_{F_k}\} \quad (1)$$

where $E_{FILTER} \subseteq E$ and $k \leq n$, i.e. E_{FILTER} is a reduced set of E produced by the filter f_{FILTER} acting on E.

Similarly, a feature extractor can be expressed as in (2)

$$E_{FEATURE} = f_{FEATURE}(E) = \{e_{FE_0}, e_{FE_1}, e_{FE_2}, \ldots, e_{FE_p}\} \quad (2)$$

where $E_{FEATURE} \neq E$, and usually $p \neq n$, which means that feature-extracted events are new ones (usually fewer in number) produced by the algorithm as a result of an internal calculation, as for example the center of mass of an object over time.

This work presents in detail two filters: the background activity filter (**BAF**), improved from [18], and the mask filter (**MF**). It additionally presents two feature extractors: the cluster object tracker (**COT**), with improvements over [19], and the object motion detector (**OMD**); all implemented and tested on FPGA. The filters have some parameters that can be configured by the user to adjust the performance of the filter to the behavior desired. This is done to make the filter flexible for further models of retinas and other neuromorphic sensors to come.

A. BACKGROUND ACTIVITY FILTER (BAF)

Due to transistor junction leakage and parasitic photocurrents [33], the DVS pixels can produce event activity unrelated to brightness changes, which looks like image noise when the events are plotted in the form of an image histogram. This activity produces continual output that will increase power consumption and could lead to errors in the post-processing algorithms, for example for tracking small objects, where the background activity leads to phantom tracking ([34]–[36]). It is possible to filter out most of this uncorrelated activity.

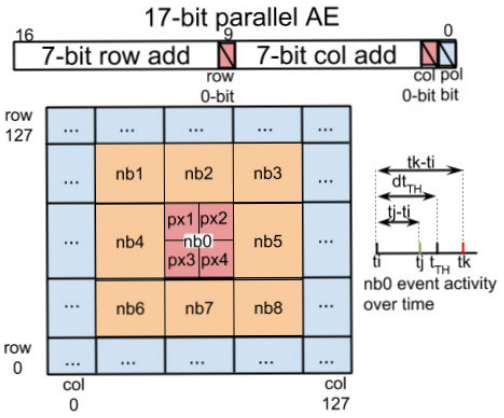


FIGURE 1. Background activity filter: Top: Address-event-representation mapping to address of 2D memory array of timestamps. Bottom-left: 2D timestamp array representation, where px1-4 corresponds to subsampling operation and nb1-nb8 represent the neighborhood of incoming address (nb0). Right: Representation of temporal filtering condition.

1) ALGORITHM

According to (1), let us call E_{BAF} the subset of E composed of those events that have limited spatio-temporal correlation. The spatial correlation is restricted to a cluster of pixel-addresses around $px1 = (a, b)$ pixel of radius nb and the temporal correlation is defined by dt_{TH} as a time interval in between event e_j and last spatially-correlated event e_i, as shown in Fig. 1., right side. For a given pixel address (a, b) and a particular time-instant t_j, let us call $e_j = (s, a, b, t_j)$ the last received event. For a neighborhood around (a, b) let us call t_i the maximum time-instant for that neighborhood that obeys $t_i < t_j$:

$$t_i = max\{t_p\} \quad (3)$$

where $t_p = time(e_p), t_p < t_j, x_p \in (a - nb, a + nb)$, $y_p \in (b-nb, b+nb)$, being nb the radix of neighbors in x-axis and y-axis considered for the filter, and (a, b) pixel is not considered. This filter says that $e_j \in E_{BAF}$, if $(t_j - t_i) \leq dt_{TH}$.

When a moving object stimulates the DVS, a neighborhood of pixels is usually active at the same time, generating events. The BAF has been further improved by also filtering events that are not correlated in space. In this way the filter removes events that are not spatio-temporally correlated. There are two possible ways to implement the spatial correlation: 1) Pixels share a position of the 2D timestamps memory array within a neighborhood (i.e. subsampling the event address by right shifting the x- and y-parts to access the 2D array of stored timestamps, px1-px4 in Fig. 1); and 2) by updating the last event timestamp in a neighborhood of timestamps around the corresponding pixel of the incoming event, avoiding its own address. So isolated pixels, with no neighbors to update their timestamps, will be filtered. 1) and 2) can be combined as in this work.

2) SOFTWARE

The jAER implementation of a background activity filter (BackgroundActivityFilter [5]) works as follows:

the timestamp of an event is used to measure the inter-event-interval times between that pixel and itself and all of its nearest neighbors (called delta-times). These delta-times (δt) are compared to a threshold to determine if this event should be filtered. If the pixel or any of its neighbors has also received an event within this time window, the event is passed along; otherwise the event is regarded as uncorrelated and is filtered out.

3) HARDWARE

Fig. 1 illustrates these two combined ideas (spatial and temporal correlations). A 2D memory of 128 × 128 timestamps is addressed taking the 7 most significant bits of the X (column) and Y (row) addresses per event. Thus, each position of the array is shared by a neighborhood of $2^n x 2^n$ pixels (e.g. 4 pixels: $px1$ to $px4$), like in [20]. Extensive spatial correlation is implemented taking more consecutive positions of the timestamps array in X and Y directions (e.g. $nb1$ to $nb8$ in the figure). This filter needs a dedicated timer to measure the inter-event-intervals. When a new event arrives, it is assumed that its address corresponds to $nb0$; the filter reads the timestamp stored in $nb0$ (called t_i) and calculates the δt by subtracting current timer value (called t_j or t_k to illustrate the two possible cases). Then, this δt is compared to the configurable threshold (called d_{TH} in the figure). If δt is lower than d_{TH} (the case of t_j) then that new event is sent out. If δt is bigger than d_{TH} (t_k case) then the event is filtered. Finally, the filter stores the current timer value (t_j for case 1 or t_k for case 2) in a set of neighbors around $nb0$, except $nb0$.

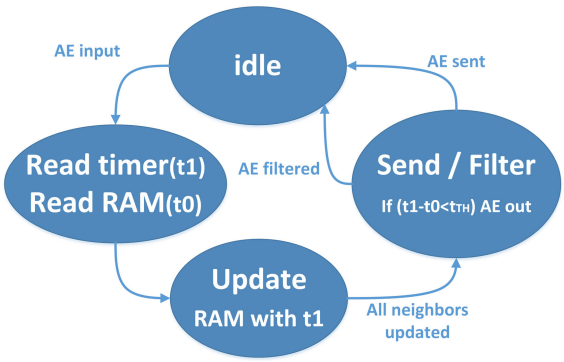


FIGURE 2. Simplified BAF FSM diagram. Temporal correlation is implemented using a global timer from which t0 and t1 are taken. Spatial correlation is implemented by UPDATE RAM state through needed iterations to cover all the neighbors around the incoming event.

This filter, implemented in VHDL, uses three main circuit components: (a) a FSM to implement the control unit (see Fig. 2 for a simplified version), (b) FPGA embedded block-RAM memory to implement the 2D array of timestamps and (c) a counter for time control (called Timer in Fig. 2).

The FSM starts with the *idle* state. If a new AER event arrives (Request signal of AER bus is active), the global timer is read and its current value is stored in a register called $t1$. At the same time (same state and clock cycle), the input event address is stored in a register (dir). In the next clock cycle, the last timestamp is read from address dir of the block-RAM. That stored value is copied into the register $t0$. Then, in the following clock cycles (8 if up to 8 neighbors $nb1 - nb8$ are used), the state machine accesses the block-RAM for writing $t1$ in the 8 neighbors addresses (except $nb0$). The $nb0$ position, if updated, must be done when a new event arrives for any of its neighbors. Once, all the neighbors' positions in memory have been updated with the new timestamp $t1$, the state machine checks if $t1 - t0$ is bigger or smaller than a configurable threshold, dt_{TH}, in order to decide if the current incoming event must be filtered or passed through to next event-processing blocks. Those 16 kwords of stored timestamps, requires a memory size of 64 kbytes. Xilinx Spartan 6 FPGAs have 36 kbit block-RAMs, which can be used as 1k blocks of 36-bit; so 16 blocks are needed. For Lattice EC3P, each embedded block RAM (EBR) has a size of 18 kbits that can be used as a 512 elements of 36 bits, so 32 EBRs are needed for this vendor. The timer is implemented using a 32-bit register that is incremented on every clock cycle.

B. MASK FILTER (MF)

There are two different situations that can be solved by using MF: 1) Filtering a set of so-called *hot pixels*; those which due to transistor mismatch have low temporal contrast thresholds and thus high spontaneous noise activity. 2) Masking out uninformative or distractor pixels, e.g. as in the slot car racer demonstration in [36], where only the pixels corresponding to the slot car race track should contribute to tracking the slot car.

1) ALGORITHM

Let us suppose an array M with the same size array as the silicon retina (AxB). This array will contain a bit at each position: $M = [m_{a,b}], m_{a,b} \in \{0, 1\}, 0 \leq a \leq A, 0 \leq b \leq B$. This bit will be understood as a mask to filter each of the incoming events from the retina location. Therefore, E_{MASK} is defined as in (4).

$$E_{MASK} = \{e_{M_0}, e_{M_1}, e_{M_2}, \ldots, e_{M_q}\}, //\forall e_{M_r} \mid M(x_{M_r}, y_{M_r}) = 1 \tag{4}$$

The M array is calculated beforehand by monitoring events for a period of time. If the frequency of events for a particular address is higher than a threshold, the mask can be activated for that address. In this case, the filter will be working as a high-pass filter. In contrast, if the mask is inverted, when the frequency is higher than the threshold, the behavior of the filter will be as a low-pass filter.

2) SOFTWARE

The jAER MF (*HotPixelFilter*) filter is implemented in two stages: an observation step and after that, the filtering step. During the observation step, a list of pixel addresses whose activity has a higher event rate than a configurable threshold is generated. Using this list, the next step is to allow only

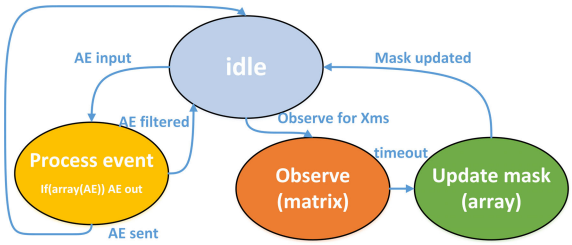


FIGURE 3. Mask Filter FSM simplified diagram. *array(ae)* is the filtering flag for the incoming event. *Observe* is a configuration signal that enables the observing state for a period of *Xms*. *timeout* indicates that the *Observing time* is due. *Matrix* is a memory used for the observing stage to collect a histogram of events. In *Update mask* each position of *array* is set if the same position of *matrix* is bigger than a configurable threshold.

the events from those pixels that are not on the list to be communicated to the next block.

3) HARDWARE

for the implementation of MF a block-RAM for the observing step ($256 \times 256 \times 5bits$, called matrix in Fig. 3) and a second and smaller block-RAM for the filtering step ($256 \times 256 \times 1bit$, called array) are used. They will be read per each incoming event to check if the event has to be filtered or passed through. A FSM (see Fig. 3 for a simplified diagram) takes care of the observing and filtering stages of the MF. From the *idle* state (blue), it is possible to advance to the *observe* stage or to process an incoming event in the normal stage (yellow one), when the observation is activated for a configurable time in *ms*. During the observation time, each incoming event is used to increment (by one) its corresponding position of the $256 \times 256 \times 5bit$ matrix. When the algorithm starts, the matrix is empty (each address content is zero). After the *Observing time*, the matrix has stored a histogram of the incoming traffic during this time. Then, the state machine evolves to a loop (green state) where the array of $256 \times 256 \times 1bit$ is updated using a configurable *Threshold*. The matrix is scanned and each 5-bit value is compared to the *Threshold*. This filter has been implemented in such a way that it is possible to invert the polarity of the flag (from the *Array*) during the normal operation.

C. CLUSTER OBJECT TRACKER (COT)

Deep learning classifiers and robotic controllers can be more accurate, effective, faster and therefore, lighter in resources and power consumption if the event-based information coming out from sensors is pre-filtered.

When using a DVS instead of a digital camera, information (packetized into frames from digital cameras) is replaced by a stream of events from DVS retinas. Each event indicates that the processing is to advance to the next step of the algorithm. Output events can be produced after processing one incoming event, which means that all the layers can work in a pipeline in a pseudo-simultaneous way [37].

1) ALGORITHM

A COT has to detect potential objects from DVS output and then follows that object while it is moving through the visual field. It can be seen that a cluster (square part of the visual field) is detected when a configurable number of events are correlated both in time and space. To allow multiple objects detection and tracking, each tracker has to work with a reduced part of the visual field, which is called *cluster*. None of these clusters can work with overlapped space addresses. When a DVS senses a moving object, events sent are very close in time and their x and y addresses use to be close in space, because they belong to an object. If we focus the attention on a particular moving object, it is possible to filter all the activity not related to that object. Furthermore this filtered activity can be mapped to the center of a new reduced visual field that can be used as an input to a next processing layer in a more accurate way, like a ConvNet classifier.

The tracker can be modeled in a similar way as in eq. (1). Nevertheless, in this work, a tracker is modeled as a feature extractor, where the center of mass (CM) of the cluster activity is calculated and continuously sent out as a new stream of events. A tracker can be expressed mathematically as in Eqs. (5)-(7):

$$E_{TRACKER} = E_{FC} \cup E_{CM} \tag{5}$$

where E_{FC} is the *filter cluster* operation and E_{CM} is the *center of mass* operation. A cluster is understood as an squared part of the visual field around the tracked object. E_{FC} and E_{CM} are formally expressed as (6) and (7):

$$\begin{aligned} E_{FC} &= f_{FilterCluster}(E) = \{e_{FC_0}, e_{FC_1}, \ldots, e_{FC_s}\}, \\ E_{FC} &\subseteq E, \quad \forall e_{FC_s} = e_i \mid (x_{CM} - R_C \\ &\qquad \leq x_i \leq x_{CM} + R_C; \\ y_{CM} - R_{CM} &\leq y_i \leq y_{CM} + R_C) \end{aligned} \tag{6}$$

$$\begin{aligned} E_{CM} &= f_{CenterMass}(E_{FC}) \\ &= \{e_{CM_0}, e_{CM_1}, \ldots, e_{CM_r}\} \\ &\qquad \forall e_{CM_r} \mid (x_{CMr} = \alpha x_{AV_s} + (1-\alpha) x_{CMr-1}; \\ &\qquad y_{CMr} = \alpha y_{AV_s} + (1-\alpha) y_{CMr-1}) \\ &\qquad \forall e_{AV_s} \mid (x_{AVs} = \alpha x_{FC_s} + (1-\alpha) x_{AVs-1}; \\ &\qquad y_{AVs} = \alpha y_{FC_s} + (1-\alpha) y_{AVs-1}) \end{aligned} \tag{7}$$

where E_{FC} are the events from E (DVS output) that fall inside the cluster, (x_{AV}, y_{AV}) are the averaged center of mass of last received events inside the cluster that are (x_{FC}, y_{FC}), E_{CM} is a set of events that represent the smooth CM of the cluster over the history, (x_{CM}, y_{CM}) represents the current center, RC is the cluster radius (half of a square side) and α is a mixing factor.

2) SOFTWARE

The cluster object tracker algorithm is implemented in jAER as the RectangularClusterTracker (**RCT**) proposed in [11] and used in [38]. This algorithm processes event packets from a DVS sensor as follows:

1) For each event (of a packet), it finds a cluster that contains the event, based on a distance criterion (like R_C in (6)). If a cluster exists, the cluster parameters (location and velocity[3]) are updated using a mixing factor ($\alpha \approx 0.01$), as expressed in Eq. (8):

$$x_{n+1} = (1-\alpha)x_n + \alpha e \tag{8}$$

where x_{n+1} is the updated location, x_n is the old location and e is the current event.

2) If the last incoming event does not fall in any cluster, then a new cluster is inferred at this event location. This new cluster will be *visible* in jAER after it has received a configurable number of events (typically 30 events).

3) After all the events in a packet are processed according to steps 1 and 2, the algorithm processes all the clusters sequentially in the following way: (a) If a cluster does not receive any new event for a configurable time, this cluster is removed. (b) If two clusters have overlapping visual fields, they are merged into one new cluster. The new cluster location is computed by averaging the locations of the two clusters. This average is weighted according to the number of events accumulated by each cluster.

3) HARDWARE

we propose a hardware COTS where each of the multiple trackers should is initialized to wait for an object at different initial locations and cluster sizes. As soon as a number of events, N_{ev}, fall into the cluster within a configurable period of time, the object has been detected. A configurable extension over the cluster size is always monitored by the tracker for dynamic decision-making on cluster movements and cluster size updates. N_{ev} can be adjusted dynamically for automatic adaptation to different object speeds and sizes as it is commented in following epigraphs.

a: CASCADING CONNECTIVITY

This tracker can be replicated as many times as necessary on an ASIC or a hardware reconfigurable device (i.e. FPGA), using the cascaded connection as represented in Fig. 4. The input stream goes to the first CM cell (**CMCell**). This unit splits the input stream in two different streams: *(1)* all the events for a detected object, called *Cluster events* in the figure, and *(2)* the rest, called *Pass Through events* in the figure. Furthermore, this cell produces a third port *(3)*, called *CM events*. This third port represents a feature extracted from the input stream that corresponds to the CM of the detected object over time. Port number 2 (*Pass Through events*) sends out all the events not falling into the cluster of the current tracker, so the output of this port represents the output of the DVS where all the events of the first detected object have been filtered. This output can be used by a next tracker for detecting a different object in the visual field. The number of trackers that can be implemented depends on the available hardware resources of the FPGA (i.e. Slices).

[3]This velocity of the software version of RCT is related to the latency between cluster location changes. It is an output of the cluster.

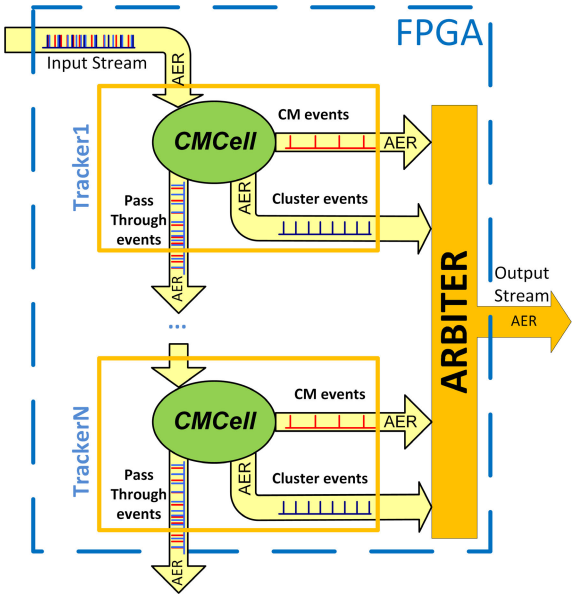


FIGURE 4. Object Trackers connected in cascade and sharing an arbiter output stage. The *pass through* port sends events that are not used. *CM* events are the center of mass of detected objects. Cluster events are those events over time detected as an object. All the buses are parallel Address-Event-Representation.

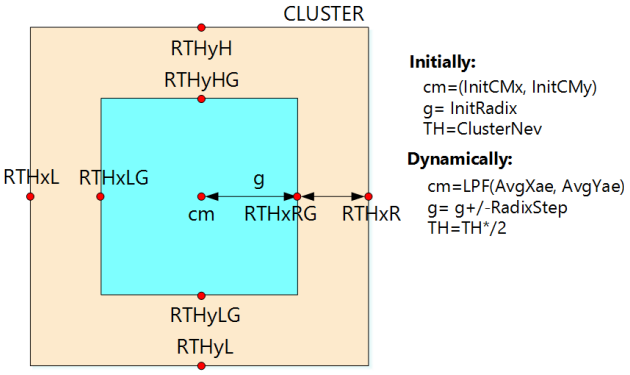


FIGURE 5. Cluster Parameters: *cm* is the updated center of mass that has a configurable initial value (*InitCMx*, *InitCMy*) and a dynamic updating with a low-pass-filter over time function *LPF*(*AvgXae*, *AvgYae*); *g* is the half-width of the cluster with a configurable initial value (*InitRadix*) and a dynamic updating of the events flow activity ($g = g + / - RadixStep$) with a configurable step; *TH* is the threshold of accumulated events in the cell that makes it fire an output event indicating the center of mass. *TH* has an initial configurable value (*ClustNev*) and it also adapts itself dynamically depending on the speed of the events ($TH = TH * /2$) by a factor of 2; *RadixTX* is the extension of the cluster. Its size is fixed and it is used for deciding about the need of increasing or decreasing the current cluster size (cyan square).

b: EVENT COUNT CONSIDERATIONS

Thanks to the event-based representation of the visual information, it is possible to have an updated calculation of the CM for each incoming event from the sensor. However, since the event-based information corresponds to a dynamically moving visual stimulus, it is necessary to compromise on the number of events (N_{ev}) used for the CM calculation (represented as **TH** in Fig. 5). If N_{ev} is low, these events can

vaguely represent the contour of the object, so the updated CM could be imprecise. In contrast, if N_{ev} is large, then the current object location could be blurred. So N_{ev} must be precisely adjusted for each particular scenario. Object speed also affects inversely the N_{ev} parameter: If the object is moving slowly, it is preferable to collect more events to obtain a more precise CM calculation. But if the object movement is faster, then N_{ev} must be decreased in order to have a good number of events that represent the current object location without blurring. In this paper we present a dynamic adjustment implementation for the N_{ev} parameter in such a way that for each CM output, after processing N_{ev} events, and taking into account the time difference between these N_{ev} events for two consecutive CM calculations, N_{ev} can be incremented or decremented.

c: SIZE MANAGEMENT

Another important issue around this object tracker is the size of the object during approach. Suppose that we have a fixed cluster size and that the detected object is approaching our center of reference (our DVS sensor). In the beginning, the object activates a small set of pixels in the DVS, but in the end the object can activate the entire DVS visual field, exceding the cluster.

In this novel hardware implementation, instead of having a fixed cluster size, we have implemented an adaptive algorithm that is able to change the size of the cluster dynamically. This algorithm takes into account an extension over the cluster size (orange area of Fig. 5). If there are events in the extension of the cluster, it means that the cluster size is too small and its size needs to be increased for a better detection of an object. And, in contrast, if all the events are falling in the cluster and no events fall in the cluster extension, then it means that the object is small and the cluster size can be reduced for the next iteration of the algorithm. As shown in Fig 5, the cluster is the portion of the visual field in between the limits ($RTHxLG$, $RTHxRG$) for x-axis and ($RTHyLG$, $RTHyHG$) for y-axis. These limits are updated every time the state machine of the tracker is in the idle state. They represent a square which center is the latest calculated CM and a $radix/2$ composed of g and RTH (radix threshold as $RadixTH$ in Table 1), where g is the $radix/2$ of the inner cluster (blue in the figure) and RTH is the space in between the inner and outer clusters (orange are in Fig. 5). Both g and RTH have an initial value, but while RTH is static, g can grow or shrink over time accordingly.

d: CM COMPUTATION

If N_{ev} is large the time in between two calculations of the CM could require a long latency, and it could also require a large memory to store the events for the CM calculation. To avoid these problems, in this work any memory or buffer is used to store events for CM calculations. The average is made following Eq. (9), which represents the average value of the x-location, implemented in the tracker state machine.

$$\overline{X_i} = \alpha X + (1-\alpha)\overline{X_{i-1}} \tag{9}$$

where $\overline{X_i}$ is the averaged x-address of the incoming events, X is the current x-address of the new event, $\overline{X_{i-1}}$ is the last averaged calculation, i takes values from 1 to N_{ev}, and α is the mixing factor. The y-location is computed similarly. In our hardware implementation, we have assumed $\alpha = 1/2^n$, where n is a configurable parameter that increase or decrease the number of last received event in the average. In this case, all the division and multiplication operations can be replaced by shift operations over the register that contains these variables. When all the N_{ev} events have been received and processed, the couple ($\overline{X_i}$, $\overline{Y_i}$) address is up to date. This average represents directly the CM.

e: LIFETIME MANAGEMENT

If a cluster, for any reason, stops receiving events from the sensor, it will be isolated in one region of the visual field. If the current detected object leaves the visual field of the sensor, or if the detected object was not a real object, the cluster will be isolated. To avoid this situation, a timer is able to reset the cluster tracker to its initial parameters. Every time the tracker's cluster receives an event, this timer is reset. If the timer overflows (after MAX clock cycles as commented in Table 1), the cluster tracker is reset to initial parameters.

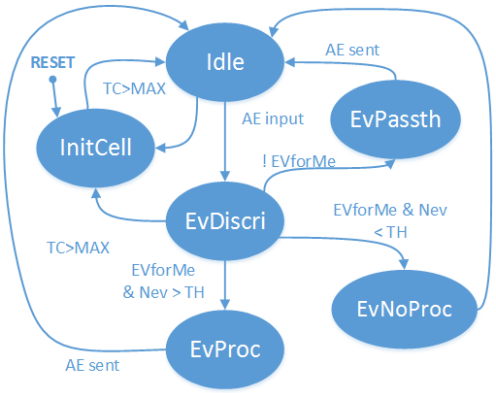


FIGURE 6. Simplified tracker FSM diagram.

f: COT FSM

Fig. 6 represents the simplified FSM diagram. *InitCell* state is the initial state after an asynchronous reset. In this state all the cluster parameters (shown in Table 1) are initialized with the values that come from the software interface (as shown in Table 3 in section V.C.1). Then, the state machine gets to the idle state to wait for incoming events. In parallel, there are two timers running (TC and $TC2$). TC manages the reset of the cluster tracker because of the absence of incoming events. $TC2$ measures the needed time for collecting N_{ev} events. The previous needed time is compared to the current one to decide if N_{ev} must be incremented or decremented

TABLE 1. Cluster object tracker parameters.

Symbol	Parameter description
InitCMx	X axis of initial CM (center cluster Y position)
InitCMy	Y axis of initial CM (center cluster Y position)
InitRadix	Initial dynamic component of cluster radix (g) that corresponds to internal reduced cluster radix
RadixTH	Static component of cluster radix. Cluster Radix is the addition of InitRadix plus RadixTH
RadixStep	Growing or Shrinking steps of dynamic cluster radix component (g)
RadixMin	Minimum allowed value for dynamic cluster radix (g)
RadixMax	Maximum allowed value for dynamic cluster radix (g)
ClustNev	Initial amount of events to be received for a CM calculation. In Fig. 5, TH represents N_{ev}, which represents the dynamic evolution of this parameter.
MAX	Number of clock cycles without receiving events inside the cluster to be wait before resetting the cluster tracker. Called *RSTtimer* in the software interface.
HistAvg	Number of past values to calculate an average before sending the next CM calculation. Needed in update(CM)

dynamically by a factor of 2, as expressed in Fig. 5 with *TH* parameter. When a new event arrives, the state machine acknowledges and captures the address. Then it goes to the *EvDiscri* state in order to discriminate if the event falls inside the current cluster size or not. If the event does not fall in the cluster size, the state machine goes to the *EvPassTh* state where the event is sent out using the *Pass Through* port, and idle state is reached again to wait for next event. In contrast, if the received event fell in the cluster, then it depends on how many events have already been received in the cluster in order to perform the average calculation of the events in the cluster, or the calculation of the next center of mass (*CM*). If current received event does not sum N_{ev} events, then the FSM goes to EvNoProc state. In this state, the averaged X and Y addresses ($\overline{X_i}$, $\overline{Y_i}$) for the last received events is updated using Eq. (9) with $\alpha = 1/2^n$ as history (in order to use shift register operations instead of multiplications and divisions). Therefore, the averaged X and Y addresses are not taking into account all N_{ev} events, but the state machine waits for them before performing the next calculations.

When the current number of received events inside the cluster is exactly N_{ev}, then the state machine moves to the *EvProc* state. In this state, g (the dynamic radius cluster) is updated taking into account the absence or presence of events in between the cluster radius and the internal sub cluster, as commented above (orange region of Fig. 5). In this state it is possible to return to the *InitCell* state if the time since the last *CM* calculation was long (*TC* overflows). In the other case, the *CM* is calculated and the number of events (N_{ev}) is updated according to those dynamic properties commented. Then an *CM* event is sent and the state machine will come back to the idle state.

D. OBJECT MOTION DETECTOR (OMD)

The *OMD* is an implementation of the so-called Object Motion Sensitive cell (**OMS**) found in the retina and reported in [39] and [40]. This particular type of Retinal Ganglion Cell (**RGC**) is excited by small objects moving in its receptive field (**RF**) and is inhibited by synchronous saccadic motion (global motion) in its surround. Preliminary results for a software and FPGA implementation of this RGC are presented in [6]. The implemented jAER model (eu.visualize.ini.retinamodel.OMCOD class in [5]) relies on a simplified form of the model in [39].

1) ALGORITHM

Fig. 7 illustrates the OMD algorithm. The *RF* of the *OMD* is composed of equally-sized subunits, representing the bipolar cells of the retina. The four central ones (see Fig. 7, left, red cells) have positive weight and contribute to the excitation of the OMD and all the surrounding ones have negative weights and contribute to its inhibition (Fig. 7, left, blue cells).

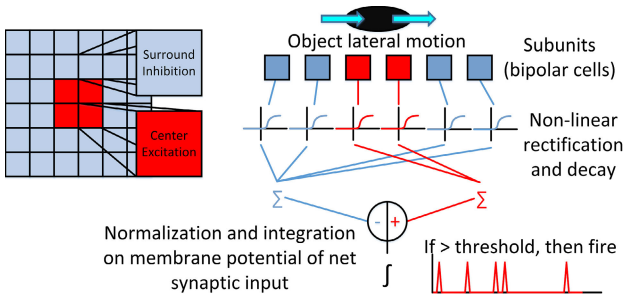


FIGURE 7. Object motion senstive cell's simplified computation. Figure adapted from [6].

When an event e is received by a subunit, its membrane voltage, V_{ex} in the case of excitatory or V_{in} in the case of inhibitory, is increased linearly. The address of the event e is assigned to a particular subunit by subsampling its address. Depending on the number of least significant bits which are ignored, the size in DVS pixels of the subunits can be increased by a power of two. Note that the polarity (s) of the DVS event e is ignored because the *OMD* performs its computation only on the energy of the movement detected and not on its brightness change sign.

While these subunits integrate linearly over time, they also exponentially decay over time with an adjustable time constant τ_s (typically a few *ms*) to adapt to the current visual situation. A non-linear rectification with saturation then transforms this voltage value. The four non-linearly transformed subunits belonging to the excitation center voltage are then summed together and normalized by their total number. The same is done for the inhibiting subunits. This way, overall normalized inhibition and excitation are calculated and their difference is computed to calculate the synaptic input. The latter simply adds to the integrate and fire (*IF*) neuron's membrane potential *Vm* which dictates whether the *OMD* shall fire or not. The decision is taken with the comparison of the threshold V_{IF}. The total normalized excitation can be scaled by a synaptic weight α which can ensure stability and can allow the *OMD* to be adjusted to different visual scenes. Finally, the *IF* neuron membrane potential *Vm* is

also decayed exponentially with time constant τ_n (typically $30ms$) to forget its integrated potential. Overall, the cell is inhibited if there is a synchronous motion in the inhibitory surround which cancels center excitation. If the excitatory subunits are activated (in the case of a small moving object) and not compensated by inhibition, then the cell fires. The computation of the *OMD* can be modelled by (10) and (11).

$$\begin{cases} if & V_m e^{-t/\tau_n} \leq V_{IF} \text{ do not fire} \\ if & V_m e^{-t/\tau_n} > V_{IF} \text{ fire} \end{cases} \tag{10}$$

$$Vm = \int_0^t \left(\alpha \frac{\sum_{i=1}^{4} \tanh(V_{ex_i} e^{\frac{-t}{\tau_s}})}{4} - \frac{\sum_{i=1}^{k-4} \tanh(V_{in_i} e^{\frac{-t}{\tau_s}})}{k-4} \right) dt \tag{11}$$

where V_{ex_i} and V_{in_i} are the i'th excitatory and inhibitory subunit membrane voltages respectively, k is the total number of subunits, τ_n is the time constant of the integrate and fire neuron and τ_s is the time constant of the subunits.

2) SOFTWARE

In [6] an array of 16 × 16 subunits were programmed in jAER. By sliding by one subunit the excitation center of the *OMD* across all subunits in the field of view, 15 × 15 *OMDs* were constructed. In the effort to reduce computation costs, all subunits, including the central ones, were considered as part of the common inhibiting surround to every *OMD* and therefore need to be computed only once. This turns the term $k - 4$ in the right hand side of (11) into just k. The result was that all overlapping *OMDs* respond to object motion at specific locations. The single *OMD* spikes in the presence of a moving object and can be easily clustered in time and space to form trackers, as presented in [6].

3) HARDWARE

With a total of 8 × 8 subunits, nine *OMDs* were implemented in the center of the nine quadrants to obtain the most basic directions about object motion. The hardware implementation differs from the jAER design because of its dual-layer structure: a Mother Cell (**MC**), which computes global inhibition shared among all *OMDs* and deals with the four-phase AER handshake protocol with the outside FSMs; and nine inner Daughter Cells (**DC**) which, in parallel, each calculate the specific center excitation. The *MC* forwards the incoming AER request and data signals to the *DCs* if the input event *e* falls in their excitation center. The *DCs* then compute whether the *OMDs* should fire or not, and the *MC* collects and sends out their output. The *DC* and *MC* therefore work together and their FSMs (shown in Fig. 8) are co-dependent.

An 8 × 8 array of 16-bit registers called *SubL* is set up in the *MC* and stores the linearly integrating membrane potential of the subunits. The array is updated for every incoming event and at every decay operation. This global decay operation (Decay state of *MC*) of the subunits is regulated by a high-priority counter (STD) with software-configurable τ_s.

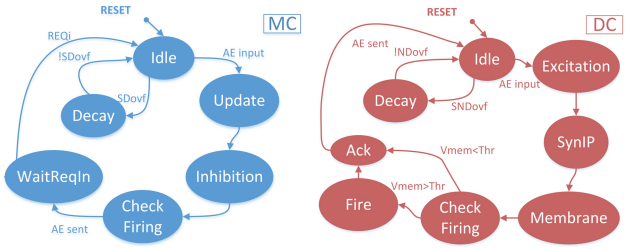


FIGURE 8. OMD FSM of Mother Cell (MC) and Daughter Cell (DC). The MC computes the global inhibition common to all OMDs and decays all subunit activity periodically. Inhibition and subunit activation are then passed to DC which computes center excitation. The DC performs integration of excitation and inhibition and computes the membrane potential of the IF neuron. The DC also computes the time decay of the IF neuron's membrane potential.

TABLE 2. Object motion detector parameters.

Symbol	Parameter description
Thr	IF neuron threshold V_{IF}.
ExW	This represents α, the synaptic weight of the excitation.
InhW	This additional factor represents the synaptic weight of the inhibition and is just used for scaling the inhibition.
SatNL	Saturation of the non-linearity.
STDLim	Limit of the counter setting the intervals at which all subunits are globally decayed.
NTDLim	Limit of the counter setting the intervals at which the OMD IF neurons are globally decayed.
TCLim	Limit of the counter used to get timestamp inside the daughter cell.

In this state, every subunit is decayed by a division by 2. This overcomes the need of a Look Up Table (LUT) for the exponential decay. If the input request signal *REQi* is active, then the FSM can proceed to *Update* state. In this state, the membrane potential subunit part of *SubL*, corresponding to the subsampled incoming event, is increased by one unit if the non-linearity that can be applied to it is still below the saturation level *SatNL* (parameters in table 2). The non-linearity is calculated and stored in another 8 × 8 array of 16-bit registers called *SubNL*. The non-linearity is implemented through a multiplying factor of 2. If the incoming event *e* falls within the excitation center of one of the *DCs*, this *DC* starts its computation.

In its next state Inhibition, the *MC* calculates its global inhibition by summing every potential in *SubNL* and storing it in *Inh*. This value is then simultaneously divided by the total number of subunits and multiplied by the synaptic weight *InhW*. In the following state, *CheckFiring*, the *MC* checks whether any of the *DCs* has acknowledged back, once they have finished processing. At every clock cycle the request signal of the *MC*, *REQo*, to the next FSM, is calculated as the AND operation of all the *DC*s output request signals. If the acknowledge signals of the *DCs* are ANDed and the result is '0', an output vector called *MCfire* is composed with the output bit of the nine *DCs* output bits, using one-hot coding. The *MC* remains in this state for as long as any of the following two conditions is not satisfied: either the *MC* has requested to any of the *DCs* with their respective request signal and any of them has acknowledged back; or no request

to the *DCs* was sent. If any of these two conditions is instead satisfied then the *MC* can move to its next and final state *WaitReqIn* where *ACKi*, the acknowledge signal back to the input is active and the request signals to the *DCs* are disabled. The *MC* returns to its *Idle* state only of the FSM preceding the OMD has withdrawn its request *REQi*.

Similarly, the FSM of the *DCs*, also starts from its idle state by setting the AER signals inactive, together with the active-high firing output *DCFire*. When the overflow signal of *NTD* is active, then the *DC* enters the *Decay* state where the membrane potential of the neuron gets halved.

The *DC* moves to its next state *Excitation* when an active low request from the *MC* is received. Here, the four *SubNL* subunits composing the center of excitation of the *DC*, are summed together and simultaneously divided by their total number of center subunits and multiplied by the synaptic weight *ExW*. The calculated excitation value is stored as *Ex*. In the next state (*SynIP*), the net synaptic input *NSI* is computed. In the next state, *Membrane*, the membrane potential of the neuron is updated by the term *NSI* multiplied by *DT* (period of time elpased from the last excitation received): added or subtracted depending on the net balance of excitation and inhibition received. *Vmem* is clipped at zero if the result of the subtraction is negative. *NSI* is multiplied by *DT* for the purpose of time integration. In the following state, *CheckFiring*, if *Vmem* is larger than or equal to the threshold, the next state will be *Fire*. If it is not, the *DC* skips to the state *Ack* directly to complete the AER handshake. In *Fire*, the firing output becomes active and the request signal of the *DC* are propagated to the next FSM. Then the FSM switches to *Ack* before returning to *Idle*. Since all *DCs* work in parallel, and their firing output is stored by the *MC* in a one-hot coded vector (*MCFire*), their processing delay does not scale up with their number.

IV. TESTING PLATFORMS

To demonstrate the portability, two different platforms, developed in two different labs, have been used to test these event-based processing filters and feature extractors: the *DevBoardUSB3* [11] from the Sensors group at the Institute of NeuroInformatics from University of Zurich, and the *AER-Node* [41] from the Robotic and Technology of Computers Lab from University of Seville. The *DevBoardUSB3* is based on Lattice FPGA, while the *AER-Node* is based on Xilinx. The sensors used are the *cnmDV* and the *DAVIS* [2], which include the operational principle of the original DVS [1]. These platforms allows to use the retinas in real-time, or use recorded streams of events taken from these sensors or synthetically generated [42], [43].

A. DEVBOARDUSB3

This platform is a host platform for any event-based processing algorithm for DVS sensors suitable for FPGA that improve previous state of the art, like [44], [45]. The board, as can be seen in Fig. 9, holds a DAVIS sensor, a Lattice ECP3 FPGA with 17k logic gates, an ADC, an Inertial Motion

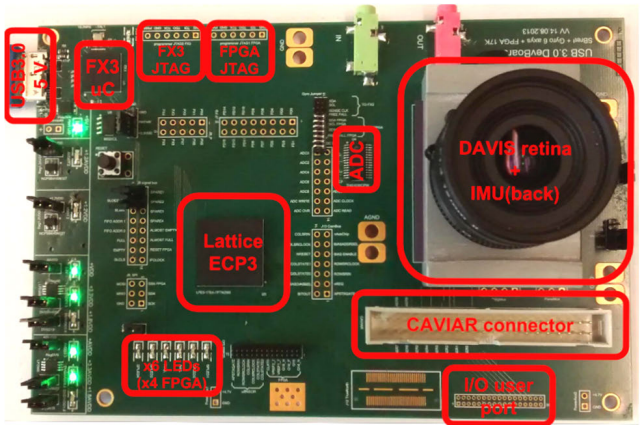


FIGURE 9. DevBoardUSB3 platform photograph. Red rectangles indicate connectors, LEDs, microcontroller, FPGA, ADC, IMU and DAVIS retina.

Unit (IMU) and a Cypress FX3 USB 3.0 Super-Speed microcontroller [19]. The IMU is used for measuring movements of the sensor [46]. It is composed of a gyroscope and an accelerometer. An onboard ADC converts the analog scan output of the sensor pixels in order to reconstruct a digital frame in the host computer. At the same time, a flow of DVS events is sent to the FPGA through a word-serial protocol. All the information is converted into events and a timestamp is assigned as needed for the data in the FPGA logic. A CAVIAR [47] connector allows to connect to this framework an external event source, like a sequencer [48]. The Cypress USB3 microcontroller is also used to configure all the parameters of the logic in the FPGA and those needed for the DAVIS sensor. It allows connectivity to TrueNorth [27].

B. AER-NODE BOARD

The design purpose of this second platform was to enable the development of mesh networks of neuromorphic chips through the expansion connectors (parallel and SATA). SATA connections are operated directly by the high-speed GTX ports of the Xilinx Spartan 6 150LXT FPGAs, to allow connectivity to other systems through serial-AER [49]. The input can come from any event-based source compatible with CAVIAR-AER buses: DVS retina, cochlea or hardware able to sequence events [48]. The event-based output of this board must be connected through the expansion connector to a monitor hardware, like *USBAERmini2* [48] or the OKAER-tool [50], which allows the log of events in a file through jAER or their real-time monitoring in the computer screen. Through a daughter board that convert USB packets into SPI commands the logic in the FPGA can be configured using either MATLAB or jAER. Fig. 10 shows a photograph of the board. It has two expansion connectors that are compatible to CAVIAR connectors but with wider buses (28-bit plus REQ/ACK). These expansion connectors can increase the functionality of the board (i.e. USB to SPI bridge).

V. RESULTS

In this work, we have developed a library of IP blocks for event-based visual processing for FPGA using VHDL.

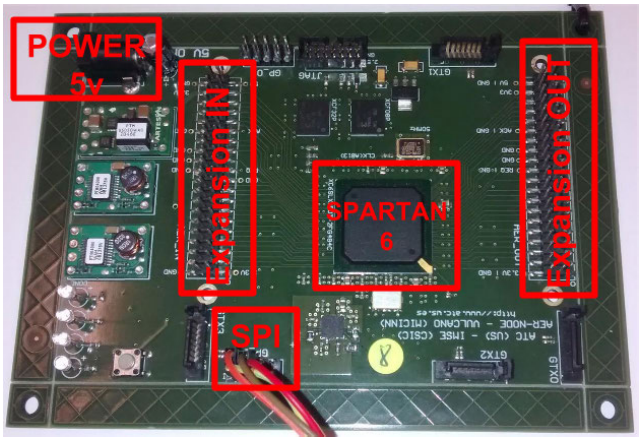


FIGURE 10. AER-Node board photograph. Red rectangles indicate expansion input and output ports for specific daughter boards, power supply and the SPI configuration bus.

A chain of these IP blocks with all the filters / feature extractors has been implemented on the two FPGA platforms. The available BLOCKRAM on these FPGAs are sufficient for the implementation of the *BAF* matrix of timestamps, the matrix of event histogram and the array of flags of the *MF*. No additional onboard components are needed except the USB to SPI bridge for the correct configuration of each filter parameters.

In this section, we present experimental results for each filter of the whole implemented chain (see Fig. 11).

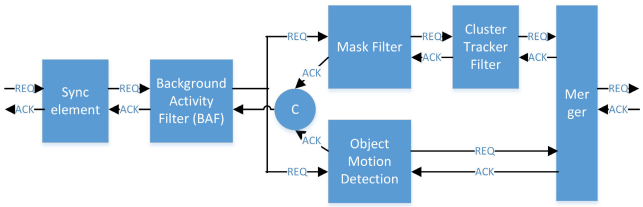


FIGURE 11. Event-based processing chain tested on the FPGA.

Since the software and hardware filter produce equivalent output, a relevant comparison is between processing time, resource cost, and power. For the software algorithm, we measure the average processing time per event using a Core-i7 4702MQ 2.2GHz laptop with 16GB RAM running Java 1.8.0. The Java implementation is optimized to minimize processing overhead and consists of single-threaded Java virtual machine instructions, which are compiled during runtime by the JIT compiler onto native CPU instructions. For the FPGA implementation, we measure the processing time per event in clock cycles times the clock frequency. On FPGA, the processing time is the same as the latency, but on a PC, there is additional latency from buffering input and output data. Previously reports show that a PC running jAER can achieve roundtrip DVS-to-USB microcontroller latencies of 2-3 ms.

Latency measurements of each block in the chain using ChipScope IPCore for the FPGA and jAER CPU time for software are shown in Table 4.

A. BACKGROUND ACTIVITY FILTER

Fig. 12. (left) shows a histogram of collected events from a DVS128 retina. As it can be seen, there are many sparse dots in the background where there are no objects. These dots correspond to sporadic activity in the pixel circuit due mostly to transistor leakage current. Fig. 12. (right) shows the same histogram after the hardware *BAF* filter is applied using a very brief δt of 100us. With this very brief correlation window, only the high contrast features of car edges are passed through and all the noise (plus a lot of signal) is filtered out.

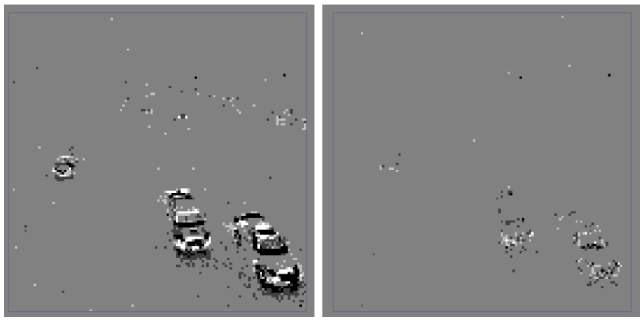


FIGURE 12. Background activity filter input (left) and hardware output (right) with a 100us δt constant.

This *BAF* implementation consumes 0.11% of the slice registers, 0.59% of the slice *LUTs* and 5.97% of available block-RAM on the Spartan 6 FPGA. The latency of this hardware per event has been measured to be 14 clock cycles with ChipScope in the *AER-Node* for a 50MHz clock, which represents 280ns as shown in Table 4.

This latency has been averaged for the jAER software version of this filter working with an AER stream coming from a hardware setup that sequences a data file through an *USBAERmini2* connected to the computer. Then, these monitored events are filtered in jAER and the software per-event processing time are averaged and shown in Table 4.

This comparision from software to hardware is fair because it ensures that input and output event timestamps are in real-time as for a DVS itself. Otherwise, if previous recorded stream of events (i.e. an aedat file) is read from a hard-disk file (including both event data and timestamps), its burst reading operation from disk are much faster than receiving these events from an AER hardware source. For the PC, the averaged jAER processing time per event for this filter is 105 ns, which is faster than the hardware because of the wait states added in the *BAF* state-machine for correct managing of read-write operations in BRAM (Fig. 4). [4]

[4]These wait states are needed for tested FPGAs because the embedded BRAM requires it. HBM memories won't require it.

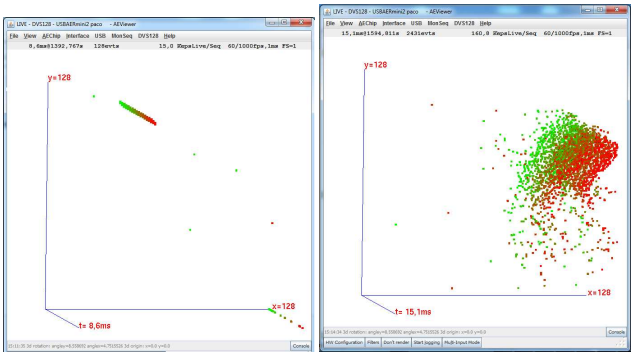


FIGURE 13. Mask filter result. Left:8.6*ms* 3D view of DVS, where top-left side visual field (128 × 128) has a damage that makes a neighborhood of 6 pixels to be hot pixels, and it has another hot pixel at address (127,0). Right side: 15*ms* 3D view of the Mask filter output showing a finger in movement over time. All hot pixels have been removed.

B. MASK FILTER

Fig. 13(left) shows a 3D space-time view of the events that represents DVS activity over time. We used a camera that was damaged during the packaging process. The result of this damage was a neighborhood of hot pixels. This sensor is useless for any post-processing if this activity remains. In the same figure (right), it can be seen the same 3D representation an active the mask filter. This *MF* implementation consumes 0,09% of the slice registers, 0,25% of the slice LUTs and 8,95% of available Block RAM on the Spartan 6 FPGA. Both jAER and FPGA latencies are shown in Table 4.

TABLE 3. BAF & COT filter parameter for experiments C.1 and C.2.

Symbol	Parameter description	Value
T_{TH}	Background Activity Filter δt	300 μs
g	Cluster dynamic radius size	5 pixels
R_{TH}	Radix Threshold for inner cluster	2 pixels
N_{ev}	Initial number of events for CM calculation	5 events (slow experiment) 3 events (fast experiment)
TC	Cluster inactivity Reset time	200ms (slow experiment) 2ms (fast experiment)
CM_h	CM history to average	4 CM events

C. CLUSTER OBJECT TRACKER

In this subsection, we show the cluster tracker output under different situations and platforms. For the two first experiments, we have used the AER-Node platform [41] and similar tracker parameters, as shown in Table 3. This COT implementation consumes 2.09% of the slice registers and 7.56% of the slice LUTs on the Spartan 6 FPGA. No Block RAM is needed. Latency comparisons between jAER and FPGA are shown in Table 4.

1) SLOW SPEED OBJECTS

Previously recorded events from a 128 × 128 DVS retina, which was set on a bridge over a 5-lane freeway monitoring

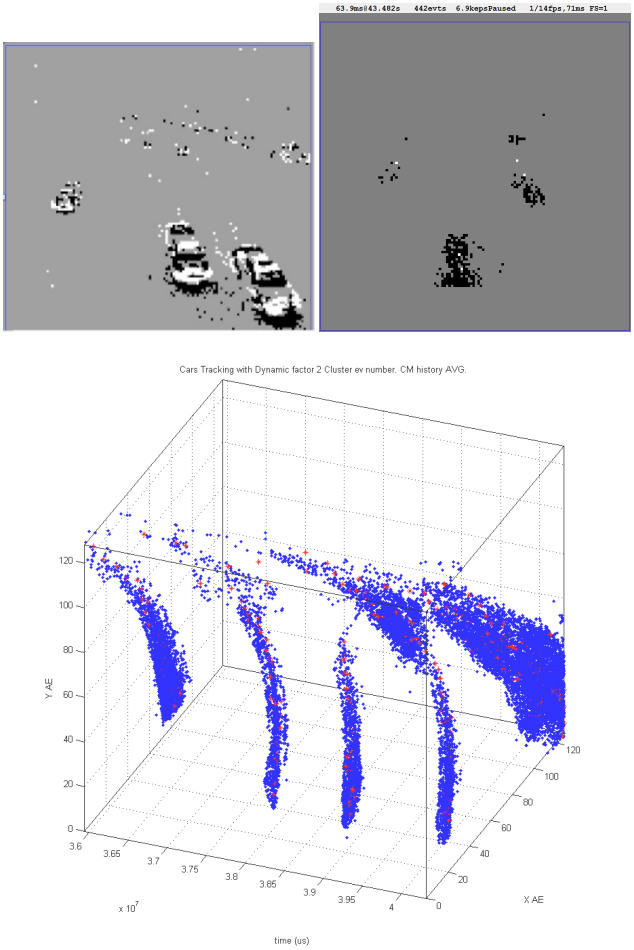


FIGURE 14. Top left: Histogram of DVS input to the system. Top right: Histogram of the output of 3 trackers: Black dots represent Cluster events and white dots represent the center of mass of the events that belong to the cluster. Bottom: 3D representations of the 4 implemented trackers output over time: Blue are cluster events and red are CM events.

many cars, have been used to test this framework for low speed objects. These recorded events can be downloaded from the INI Sensor's group web. [5]

Fig. 14 shows a histogram with the outputs of the trackers, from the same stimulus shown in Fig. 13. In this histogram, three object trackers are sending their results since only three cars are in the visual field. The bottom graph represents a 3D view of the trackers over time (cluster events in blue, and CM events in red). Once a tracker losses a car it is reset and it starts to track a new car. Fig. 15 shows the output for one tracker (one car) over 1.2 seconds. Events falling inside the clusters represent a moving car (blue). When the car is far away, only a few events are produced at the horizon line (around row address 50). The closer the car is to the bridge, the bigger it becomes, and so, the more events are produced by the DVS sensor (lower row addresses). Idle clusters are reset and used again to track new objects.

[5]http://sensors.ini.uzh.ch/databases.html

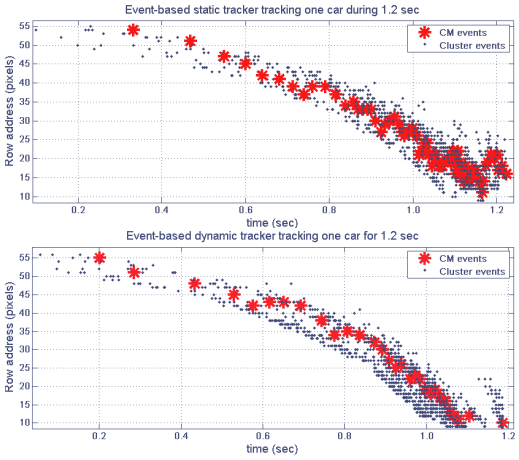


FIGURE 15. Car tracking using BAF and x4 trackers. Only 1 tracker output is shown. Top: Without dynamic adaptation. Bottom: With dynamic adaptation. Blue dots represent events that fall in the cluster. Red dots represent CM events.

Fig. 15 (top) represents the output of a tracker without dynamic adaptation of N_{ev}. When the car is closer (so bigger and faster motion), CM output is not precise. Fig. 15 (bottom) shows the output with dynamic capabilities. N_{ev} is increased at the same time the car is approaching, and the CM output is more precise. CM inter event interval time is 150ms when the car is far and 10ms when the car is closer. N_{ev} is fixed to 15 events for the static version and it was initialized to 5 in the dynamic one, where it oscillates between 10 and 80 depending on the distance of the car to the DVS.

2) HIGH SPEED OBJECTS

In this experiment, a set of 52 poker cards was riffled in front of the DVS sensor in 500 ms, i.e. at less than 10 ms per card. Regular frame-based digital cameras cannot capture proper images to be processed in this scenario unless they use a frame rate of over 200Hz, as stated in [37]. With DVS events, no special operation is needed. The same cluster radius and extension size tracker parameters (Table 1) are used in both experiments. Fig 16 (top-left) is a histogram of events produced by the falling cards in front of a DVS retina in the 500 ms window. On its right, the histogram of events shown represents the output of one tracker for the same falling card. The tracker initial position was configured for that column shape to the top side of the visual field. The bottom part of the figure shows the whole output of the tracker for the falling card from top to bottom of the retina visual field (16ms ms). At the beginning the card is still and when the finger releases the card, it speeds up. It can be seen how the number of events inside the cluster per time unit keep growing over time (blue dots), while number of CM events per time unit starts to adjust itself to the increment of speed and stays in the same range (same throughput). In the beginning, the CM inter-event interval is 750 μs, and then it is adjusted at 25 μs. N_{ev} has an initial value of 3 events and it grows up to 12 to accommodate the stimulus.

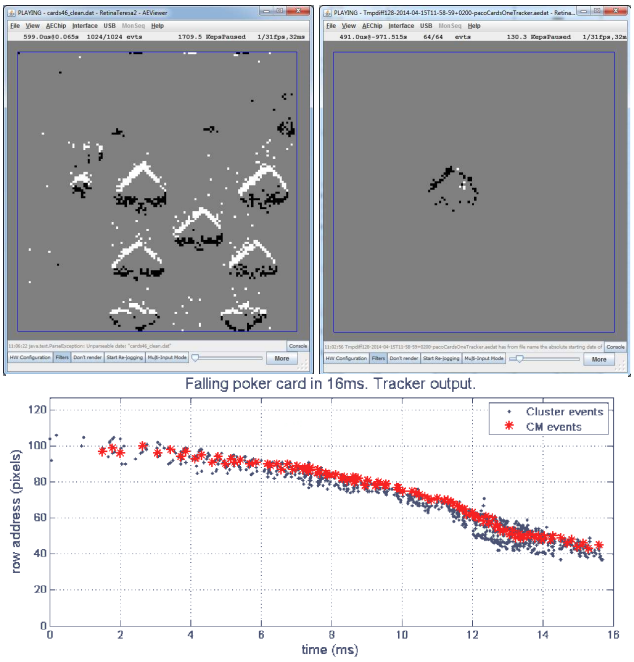


FIGURE 16. Poker card fast tracking for 16ms with a Dynamic tracker. Top-left represents a histogram of a poker card for 600us. Top-right is a histogram of the tracker output for the same card (500us). Bottom is the whole tracking.

3) SPINNING DOT TRACKING

With the DevBoardUSB3 platform, we have synthesized and implemented in the FPGA an event-based processing chain that includes the BAF, the MF and the COT. It also includes the logic that enables this platform to work properly with the DAVIS DVS events and the USB3 interface. The monitoring logic for the FPGA [19] joins several state machines that take care of different kinds of information and then a multiplexer state machine that takes care of the timestamps assignment for each kind of event. The communication between different states machines is done by using small FIFOs. All collected traffic in the output of the multiplexer state machine is sent to the FIFO of the Cypress FX3 microcontroller, so they are collected in jAER through a USB3.0 super-speed interface.

With this logic, events arriving to jAER can be divided in different categories in order to assign different colors to different filter outputs. Fig. 17 (top) shows a 3D graph with a temporal representation of the cluster tracker output when the system is detecting and tracking a 1cm diameter dark circle in a small white cupboard disc attached to a hand fan that makes the circle to spin at 100rpm, so each spin is completed in approximately 1ms. Red dots represent the events that fall into the cluster, and blue dots correspond to the center of mass of the circle over time. In this same figure (bottom), it is also shown a screenshot of the jAER while it is monitoring both the DAVIS output and the tracker output sharing the same USB3.

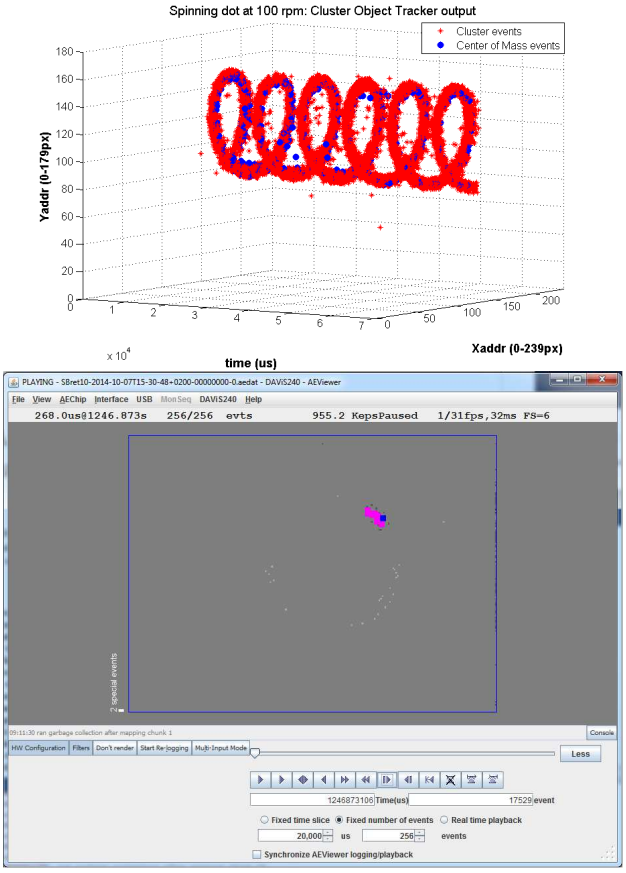


FIGURE 17. Spinning a disk with a black 1 cm circle at 100rpm. Top: 3D temporal evolution graph of the cluster object tracker output and a photograph of the stimulus generator. Bottom: A screenshot of the jAER DVS spikes binned over 260μs, with cluster tracker events (pink) and center of mass events (blue).

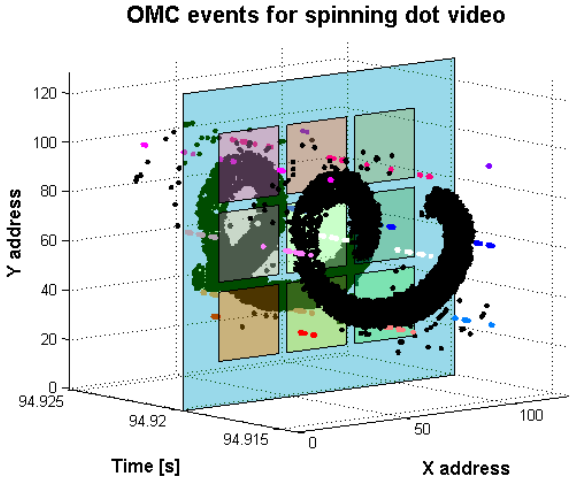


FIGURE 18. Nine OMDs firing in time when the spinning dot from Fig. 17 excites them. Spinning-dot flow of events in black, and OMD output events in different colors.

D. OBJECT MOTION DETECTION

The output of the OMD with 9 DCs is shown in Fig. 18. Each DC is active when the spinning dot is inside its region. This OMD implementation consumes 5% of the slice registers, 12% of the slice LUTs and occupies 15% of the available DSP slices on the Spartan 6 FPGA. Furthermore, no complex components are required in the synthesis of the OMDs. Using the on-chip logic analyzer provided by Xilinx (ChipScope) synthesized together with the OMDs, the processing delay of the OMD was measured to be between 11 and 22 clock cycles, below the microsecond event timestamping resolution of the retina. This delay variation depends on the incoming event e falling in one of the DC's receptive fields, what requires more computation. With a 50 MHz clock this corresponds to a 220 and 440 ns delay respectively. Table 4 shows a latency versus processing-time comparison between FPGA and jAER, where the speedup percentage of improvement has been calculated according to Amdahl's law (Eq. (12)) [51]:

TABLE 4. Filter processing time/latency comparison between JAER and FPGA.

Filter	jAER Processing Time (ns/event)	FPGA Latency (ns/event)	% improvement
Background Activity Filter	80-130	280	-166
Mask Filter	120-220	80	+112.5
Cluster Object Tracker (cluster/CM)	600-1300	120 / 140	+571
Object Motion Detector	100-1500	240	+233

$$A = \frac{t_{exe_slow}}{t_{exe_fast}} \geq 1,\ A = (1 + X),\ X = \%improvement \quad (12)$$

VI. CONCLUSION

This paper presents a set of event-based postprocessing algorithms for visual information coming from a DVS event camera. The description of the algorithms, with links to the software jAER implementations, and hardware implementation details for FPGA are presented. This work demonstrates that with simple resources (Finite State Machines composed of flip-flops, adders, comparators; and SRAMs) it is possible to implement efficient real-time event-based filters and feature extractors. Latencies of these algorithms are in the order of hundreds of nanoseconds with 50 MHz clock frequency and they are faster (up to 570%) compared with the software implementations in jAER running on a 2.2GHz i7 CPU.

More importantly, the FPGA architecure processes each event as it arrives with nearly deterministic and sub-millisecond latency, compared with a USB interface to a PC, where thread scheduling and memory buffers contribute variable latencies on the order of milliseconds [34]. These predictable and short latencies will enable closed-loop control with greater than 1kHz bandwidth, which is only possible on high-performance computers using specialized high-frame rate cameras and powerful lighting.

Although we did not measure power consumption explicitly, the FPGA with wall-plug power consumption of under 10W consumes only 1/5 of the power of a typical fully-loaded 50W laptop computer.

The filters and features extractors presented in this paper can be applied to other neuromorphic sensors such as the silicon cochlea [52], or NAS [53] and the olfactory sensor [54]. In order to benefit from presented results, they could be applied to novel event-based processing systems that use mesh networks of convolutions [28], or multiconvolutional processors [55], [56] that allow the implementation of ConvNets for event-based classification; and architectures able to implement event-based fully connected spiking deep networks, e.g. Minitaur [29].

Most of the implementations described here are available in the open-source jAER project[6] and in our Git-Hub repository.[7] We hope that this work will stimulate the development of further event-driven logic circuits for neuromorphic computing.

ACKNOWLEDGMENT

The authors would like to thank the Capocaccia Neuromorphic Cognition Workshop, iniLabs GmbH (especially Luca Longinotti and Vicente Villanueva), the University of Zurich and ETH Zurich.

REFERENCES

[1] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128 × 128 120 dB 15 μs latency asynchronous temporal contrast vision sensor," *IEEE J. Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, Mar. 2008.

[2] C. Posch, T. Serrano-Gotarredona, B. Linares-Barranco, and T. Delbruck, "Retinomorphic event-based vision sensors: Bioinspired cameras with spiking output," *Proc. IEEE*, vol. 102, no. 10, pp. 1470–1484, Oct. 2014.

[3] M. A. Sivilotti, "Wiring considerations in analog VLSI systems, with application to field-programmable networks," California Inst. Technol., Pasadena, CA, USA, 1991.

[4] K. A. Boahen, "Communicating neuronal ensembles between neuromorphic chips," in *Neuromorphic Systems Engineering*. Boston, MA, USA: Springer, 1998, pp. 229–259.

[5] *jAER Open Source Project*. Accessed: Mar. 4, 2019. [Online]. Available: https://github.com/SensorsINI/jaer

[6] D. P. Moeys, T. Delbrück, A. Rios-Navarro, and A. Linares-Barranco, "Retinal ganglion cell software and FPGA model implementation for object detection and tracking," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 1434–1437.

[7] T. Delbrück, "Frame-free dynamic digital vision," in *Proc. Int. Symp. Secure-Life Electron., Adv. Electron. Qual. Life Soc.*, 2008, pp. 21–26.

[8] S.-C. Liu, T. Delbrück, G. Indiveri, A. Whatley, and R. Douglas, Eds., *Event-Based Neuromorphic Systems*. London, U.K.: Wiley, 2015. doi: 10.1002/9781118927601.

[9] A. Lakshmi, A. Chakraborty, and C. S. Thakur, "Neuromorphic vision: From sensors to event-based algorithms," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 9, no. 4, p. e1310, 2019.

[10] G. Gallego, T. Delbrück, G. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based vision: A survey," 2019, *arXiv:1904.08405*. [Online]. Available: https://arxiv.org/abs/1904.08405

[11] T. Delbrück and P. Lichtsteiner, "Fast sensory motor control based on event-based hybrid neuromorphic-procedural system," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2007, pp. 845–848.

[12] D. R. Valeiras, X. Lagorce, X. Clady, C. Bartolozzi, S.-H. Ieng, and R. Benosman, "An asynchronous neuromorphic event-driven visual part-based shape tracking," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 12, pp. 3045–3059, Mar. 2015.

[13] D. Tedaldi, G. Gallego, E. Mueggler, and D. Scaramuzza, "Feature detection and tracking with the dynamic and active-pixel vision sensor (DAVIS)," in *Proc. 2nd Int. Conf. Event-Based Control, Commun., Signal Process. (EBCCSP)*, Jun. 2016, pp. 1–7.

[6] http://jaerproject.org

[7] https://github.com/RTC-research-group/EDIP_library

[14] I. Alzugaray and M. Chli, "Asynchronous corner detection and tracking for event cameras in real time," *IEEE Robot. Automat. Lett.*, vol. 3, no. 4, pp. 3177–3184, Oct. 2018.

[15] F. Barranco, C. Fermuller, and E. Ros, "Real-time clustering and multi-target tracking using event-based sensors," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2018, pp. 5764–5769.

[16] S.-H. Ieng, C. Posch, and R. Benosman, "Asynchronous neuromorphic event-driven image filtering," *Proc. IEEE*, vol. 102, no. 10, pp. 1485–1499, Oct. 2014.

[17] T. Serrano-Gotarredona, A. G. Andreou, and B. Linares-Barranco, "AER image filtering architecture for vision-processing systems," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 46, no. 9, pp. 1064–1071, Sep. 1999.

[18] F. Gómez-Rodríguez, L. Miró-Amarante, F. Díaz-del-Rio, A. Linares-Barranco, and G. Jimenez, "Real time multiple objects tracking based on a bio-inspired processing cascade architecture," in *Proc. IEEE Int. Symp. Circuits Syst.*, Jun. 2010, pp. 1399–1402.

[19] A. Linares-Barranco, F. Gómez-Rodríguez, V. Villanueva, L. Longinotti, and T. Delbrück, "A USB3.0 FPGA event-based filtering and tracking framework for dynamic vision sensors," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 2417–2420.

[20] H. Liu, C. Brandli, C. Li, S.-C. Liu, and T. Delbrück, "Design of a spatiotemporal correlation filter for event-based sensors," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 722–725.

[21] A. Khodamoradi and R. Kastner, "O(N)-space spatiotemporal filter for reducing noise in neuromorphic vision sensors," *IEEE Trans. Emerg. Topics Comput.*, to be published.

[22] B. Ramesh, A. Ussa, L. D. Vedova, H. Yang, and G. Orchard, "PCA-RECT: An energy-efficient object detection approach for event cameras," in *Proc. 14th Asian Conf. Comput. Vis.*, vol. 11367. Perth, WA, Australia: Springer, Dec. 2018, pp. 434–449.

[23] C. Brandli, R. Berner, M. Yang, S.-C. Liu, and T. Delbrück, "A 240 × 180 130 dB 3 μs latency global shutter spatiotemporal vision sensor," *IEEE J. Solid-State Circuits*, vol. 49, no. 10, pp. 2333–2341, Oct. 2014.

[24] B. Ramesh, H. Yang, G. M. Orchard, N. A. Le Thi, S. Zhang, and C. Xiang, "DART: Distribution aware retinal transform for event-based cameras," *IEEE Trans. Pattern Anal. Mach. Intell.*, to be published.

[25] S. Hoseini and B. Linares-Barranco, "Real-time temporal frequency detection in FPGA using event-based vision sensor," in *Proc. IEEE 14th Int. Conf. Intell. Comput. Commun. Process. (ICCP)*, Sep. 2018, pp. 271–278.

[26] V. Padala, A. Basu, and G. Orchard, "A noise filtering algorithm for event-based asynchronous change detection image sensors on truenorth and its implementation on truenorth," *Frontiers Neurosci.*, vol. 12, p. 118, Mar. 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2018.00118

[27] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1537–1557, Oct. 2015.

[28] C. Zamarreno-Ramos, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "Multicasting mesh AER: A scalable assembly approach for reconfigurable neuromorphic structured AER systems. Application to convNets," *IEEE Trans. Biomed. Circuits Syst.*, vol. 7, no. 1, pp. 82–102, Feb. 2013.

[29] D. Neil and S.-C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 12, pp. 2621–2628, Dec. 2014.

[30] F. Perez-Peña, A. Morgado-Estevez, A. Linares-Barranco, A. Jimenez-Fernandez, F. Gomez-Rodriguez, G. Jimenez-Moreno, and J. Lopez-Coronado, "Neuro-inspired spike-based motion: From dynamic vision sensor to robot motor open-loop control through spike-VITE," *Sensors*, vol. 13, no. 11, pp. 15805–15832, 2013.

[31] D. P. Moeys, F. Corradi, E. Kerr, P. Vance, G. Das, D. Neil, D. Kerr, and T. Delbrück, "Steering a predator robot using a mixed frame/event-driven convolutional neural network," in *Proc. 2nd Int. Conf. Event-Based Control, Commun., Signal Process. (EBCCSP)*, Jun. 2016, pp. 1–8.

[32] D. P. Moeys, D. Neil, F. Corradi, E. Kerr, P. Vance, G. Das, S. A. Coleman, T. M. McGinnity, D. Kerr, and T. Delbrück, "PRED18: Dataset and further experiments with DAVIS event camera in predator-prey robot chasing," 2018, *arXiv:1807.03128*. [Online]. Available: https://arxiv.org/abs/1807.03128

[33] Y. Nozaki and T. Delbrück, "Temperature and parasitic photocurrent effects in dynamic vision sensors," *IEEE Trans. Electron Devices*, vol. 64, no. 8, pp. 3239–3245, Aug. 2017.

[34] T. Delbrück and M. Lang, "Robotic goalie with 3 ms reaction time at 4% CPU load using event-based dynamic vision sensor," *Frontiers Neurosci.*, vol. 7, p. 223, Nov. 2013. [Online]. Available: http://journal.frontiersin.org/Journal/10.3389/fnins.2013.00223/full

[35] J. Conradt, M. Cook, R. Berner, P. Lichtsteiner, R. J. Douglas, and T. Delbrück, "A pencil balancing robot using a pair of AER dynamic vision sensors," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2009, pp. 781–784.

[36] T. Delbrück, M. Pfeiffer, R. Juston, G. Orchard, E. Müggler, A. Linares-Barranco, and M. W. Tilden, "Human vs. computer slot car racing using an event and frame-based DAVIS vision sensor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 2409–2412.

[37] C. Farabet, R. Paz, J. Pérez-Carrasco, C. Zamarreño-Ramos, A. Linares-Barranco, Y. LeCun, E. Culurciello, T. Serrano-Gotarredona, and B. Linares-Barranco, "Comparison between frame-constrained fix-pixel-value and frame-free spiking-dynamic-pixel ConvNets for visual processing," *Frontiers Neurosci.*, vol. 6, p. 32, Apr. 2012.

[38] M. Litzenberger, C. Posch, D. Bauer, A. Belbachir, P. Schon, B. Kohn, and H. Garn, "Embedded vision system for real-time object tracking using an asynchronous transient vision sensor," in *Proc. IEEE 12th Digit. Signal Process. Workshop, 4th IEEE Signal Process. Educ. Workshop*, Sep. 2006, pp. 173–178.

[39] B. P. Ölveczky, S. A. Baccus, and M. Meister, "Segregation of object and background motion in the retina," *Nature*, vol. 423, no. 6938, p. 401, 2003.

[40] S. A. Baccus, B. P. Ölveczky, M. Manu, and M. Meister, "A retinal circuit that computes object motion," *J. Neurosci.*, vol. 28, no. 27, pp. 6807–6817, 2008.

[41] T. Iakymchuk, A. Rosado, T. Serrano-Gotarredona, B. Linares-Barranco, A. Jiménez-Fernández, A. Linares-Barranco, and G. Jiménez-Moreno, "An AER handshake-less modular infrastructure PCB with x8 2.5 Gbps LVDS serial links," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 1556–1559.

[42] A. Linares-Barranco, G. Jimenez-Moreno, B. Linares-Barranco, and A. Civit-Balcells, "On algorithmic rate-coded AER generation," *IEEE Trans. Neural Netw.*, vol. 17, no. 3, pp. 771–788, May 2006.

[43] F. Gomez-Rodriguez, R. Paz, L. Miro, A. Linares-Barranco, G. Jimenez, and A. Civit, "Two hardware implementations of the exhaustive synthetic AER generation method," in *Computational Intelligence and Bioinspired Systems*, J. Cabestany, A. Prieto, and F. Sandoval, Eds. Berlin, Germany: Springer, 2005, pp. 534–540.

[44] F. Gomez-Rodriguez, R. Paz, A. Linares-Barranco, M. Rivas, L. Miro, S. Vicente, G. Jimenez, and A. Civit, "AER tools for communications and debugging," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2006, p. 4.

[45] M. Rivas, F. Gomez-Rodriguez, R. Paz, A. Linares-Barranco, S. Vicente, and D. Cascado, "Tools for address-event-representation communication systems and debugging," in *Proc. Int. Conf. Artif. Neural Netw.* Berlin, Germany: Springer, 2005, pp. 289–296.

[46] T. Delbrück, V. Villanueva, and L. Longinotti, "Integration of dynamic vision sensor with inertial measurement unit for electronically stabilized event-based vision," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 2636–2639.

[47] R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, L. Camuñas-Mesa, R. Berner, M. Rivas-Pérez, T. Delbrück, S.-C. Liu, R. Douglas, P. Häfliger, G. Jiménez-Moreno, A. C. Ballcels, T. Serrano-Gotarredona, A. J. Acosta-Jiménez, and B. Linares-Barranco, "CAVIAR: A 45k neuron, 5M synapse, 12G connects/s AER hardware sensory–processing–learning–actuating system for high-speed visual object recognition and tracking," *IEEE Trans. Neural Netw.*, vol. 20, no. 9, pp. 1417–1438, Sep. 2009.

[48] R. Berner, T. Delbrück, A. Civit-Balcells, and A. Linares-Barranco, "A 5meps $100 USB2.0 address-event monitor-sequencer interface," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2007, pp. 2451–2454.

[49] A. Yousefzadeh, M. Jabło ski, T. Iakymchuk, A. Linares-Barranco, A. Rosado, L. A. Plana, S. Temple, T. Serrano-Gotarredona, S. B. Furber, and B. Linares-Barranco, "On multiple AER handshaking channels over high-speed bit-serial bidirectional LVDS links with flow-control and clock-correction on commercial FPGAs for scalable neuromorphic systems," *IEEE Trans. Biomed. Circuits Syst.*, vol. 11, no. 5, pp. 1133–1147, Oct. 2017.

[50] A. Rios-Navarro, J. P. Dominguez-Morales, R. Tapiador-Morales, D. Gutierrez-Galan, A. Jimenez-Fernandez, and A. Linares-Barranco, "A 20Mevps/32Mev event-based USB framework for neuromorphic systems debugging," in *Proc. 2nd Int. Conf. Event-Based Control, Commun., Signal Process. (EBCCSP)*, Jun. 2016, pp. 1–6.

[51] D. P. Rodgers, "Improvements in multiprocessor system design," in *Proc. 12th Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Alamitos, CA, USA, 1985, pp. 225–231. [Online]. Available: http://dl.acm.org/citation.cfm?id=327010.327215

[52] S.-C. Liu, A. van Schaik, B. A. Minch, and T. Delbrück, "Asynchronous binaural spatial audition sensor with $2 \times 64 \times 4$ channel output," *IEEE Trans. Biomed. Circuits Syst.*, vol. 8, no. 4, pp. 453–464, Aug. 2014.

[53] A. Jiménez-Fernández, E. Cerezuela-Escudero, L. Miro-Amarante, M. J. Dominguez-Morales, F. de Asis Gomez-Rodriguez, A. Linares-Barranco, and G. Jimenez-Moreno, "A binaural neuromorphic auditory sensor for FPGA: A spike signal processing approach," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 4, pp. 804–818, Apr. 2017.

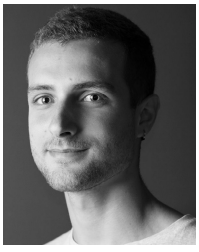
[54] T. J. Koickal, A. Hamilton, S. L. Tan, J. A. Covington, J. W. Gardner, and T. C. Pearce, "Analog VLSI circuit implementation of an adaptive neuromorphic olfaction chip," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 54, no. 1, pp. 60–73, Jan. 2007.

[55] R. Tapiador-Morales, A. Linares-Barranco, A. Jimenez-Fernandez, and G. Jimenez-Moreno, "Neuromorphic LIF row-by-row multiconvolution processor for FPGA," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 1, pp. 159–169, Feb. 2019.

[56] L. A. Camuñas Mesa, Y. L. Domínguez-Cordero, A. Linares-Barranco, T. Serrano-Gotarredona, and B. Linares-Barranco, "A configurable event-driven convolutional node with rate saturation mechanism for modular ConvNet systems implementation," *Frontiers Neurosci.*, vol. 12, p. 63, Feb. 2018. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2018.00063

ALEJANDRO LINARES-BARRANCO (M'04–SM'17) received the B.S. degree in computer engineering, the M.S. degree in industrial computer engineering, and the Ph.D. degree in computer engineering (specialized in computer interfaces for neuromorphic systems) from the University of Seville, in 1998, 2002, and 2003, respectively. He has been an Associate Professor with the University of Seville, since 2009, and the Head of the Architecture and Computer Technology Department.

FERNANDO PEREZ-PEÑA (M'04) received the degree in telecommunication engineering from the University of Seville, Spain, in 2009, and the Ph.D. degree (specialized in neuromorphic motor control) from the University of Cadiz, Spain, in 2014. In 2015, he was a Postdoctoral Researcher with CITEC, Bielefeld University, Germany. He has been an Assistant Professor with the University of Cadiz, since 2014. His current research interests include neuromorphic engineering, FPGA digital design, motor control, and neurorobotics.



DIEDERIK PAUL MOEYS received the M.Eng. degree from University College London, London, U.K., in 2013, and the Ph.D. degree in electronic engineering from ETH Zurich, Zurich, Switzerland, in 2016. His current research interest includes chip design to event-based data processing and tracking using machine learning. He received the Goldsmid Faculty Medal, in 2013, the IET Prize, and the IET Postgraduate Scholarship, in 2016.

FRANCISCO GOMEZ-RODRIGUEZ received the B.S. degree in computer engineering and the Ph.D. degree in computer science (specialized neuromorphic systems) from the University of Seville, Spain, in 1999 and 2011, respectively. Since 2003, he has been a member of the Robotics and Technology of Computers Laboratory, University of Seville. He is currently an Assistant Professor with the Computer Architecture and Technology Department, University of Seville. He participated in several EU and Spanish research project in the area of neuromorphic systems. His current research interests include vision and auditory neuromorphic systems and theirs hardware implementations.

GABRIEL JIMENEZ-MORENO received the M.S. degree in physics (electronics) and the Ph.D. degree from the University of Seville, Seville, Spain, in 1987 and 1992, respectively, where he is currently an Associate Professor of computer architecture. He participated in the creation of the Department of Computer Architecture, University of Seville. He has authored various articles and research reports on robotics, rehabilitation technology, and computer architecture.

SHIH-CHII LIU (M'02–SM'07) received the bachelor's degree in electrical engineering from the Massachusetts Institute of Technology and the Ph.D. degree in the computation and neural systems program from Caltech, in 1997. She was with various companies, including Gould American Microsystems, LSI Logic, and Rockwell International Research Labs. She is currently a Group Leader with the Institute of Neuroinformatics.

TOBI DELBRUCK (M'99–SM'06–F'13) received the B.Sc. degree in physics from UC San Diego, in 1986, and the Ph.D. degree from Caltech, in 1993. He has been with the Institute of Neuroinformatics, ETH Zurich, since 1998, where he is currently a Professor in physics and electrical engineering. His group with SC Liu focuses on neuromorphic sensory processing and efficient deep learning.